Proseminar
aus

Algorithmen und Datenstrukturen

Graph traversals - Durchlaufen von Graphen

von Kinateder Thomas

1 Einleitung

Für manche Probleme ist es wichtig, alle Knoten in einem Graphen zu betrachten. So kann man etwa einer in einem Labyrinth eingeschlossenen Person nachfühlen, daß sie gerne sämtliche Kreuzungen von Gängen in Augenschein nehmen will. Die Gänge des Labyrinths sind hier die Kanten des Graphen, und Kreuzungen von Gängen sind Knoten. Das Betrachten eines Knotens nennt man auch oft Besuchen des Knotens. Manchmal ist es wichtig, die Knoten nach einer gewissen Systematik zu besuchen. So kann man sich leicht vorstellen, daß eine einzelne Person im Labyrinth einem Gang zunächst eine Weile folgt, bevor sie vielleicht schließlich kehrt macht, also mit der Suche zunächst „in die Tiefe“ des Labyrinths geht; suchen dagegen mehrere Personen gleichzeitig, so werden sie eher vom Startpunkt aus ausschwärmen, also „in die Breite“ gehen. Im Folgenden werden die Breiten- und die Tiefensuche als die beiden Hauptvertreter der Graphdurchläufe näher betrachtet und dabei auf ihre wichtigsten Eigenschaften und Anwendungen eingegangen.

2 Breitensuche

Das Prinzip der Breitensuche (breadth-first search) ist das „wellenartige“ Ausbreiten vom Startknoten weg, d.h. die Suche beginnt beim Startknoten, danach werden die Nachbarn des Startknoten besucht, danach die Nachbarn der Nachbarn, usw. Dadurch kann die Knotenmenge - entsprechend ihrer minimalen Anzahl von Kanten zum Startknoten - in Level unterteilt werden. Im Level 0 befindet sich nur der Startknoten, das Level 1 besteht aus allen Nachbarn des Startknotens, usw.

Algorithmus Für den hier vorgestellten Algorithmus wird braucht man als Eingabe einen Graphen G = (V, E) sowie einen Startknoten s. Um Aussagen über die Struktur und die Eigenschaften des Graphen treffen zu können, speichert man zu jedem Knoten $v \in V$ einige Informationen:

- **color[v]**: repräsentiert den aktuellen Bearbeitungsstatus:
  weiß = unbesucht/unbearbeitet
  schwarz = abgearbeitet (v und alle Nachbarn von v wurden besucht)
  grau = in Bearbeitung (v wurde besucht, kann aber noch unbesuchte Nachbarn haben)

- **p[v]**: Vorgänger (=predecessor) von v

- **d[v]**: Distanz zum Startknoten bzgl. der minimalen Kantenanzahl

Zum Speichern der Knoten, die in Bearbeitung sind, wird eine Warteschlange Q (FIFO) verwendet.

BFS(G, s)

```
1    for each vertex u ∈ V[G]-{s}
2        do color[u] ← WHITE
3           d[u] ← ∞
4           p[u] ← NIL
5    color[s] ← GRAY
6    d[s] ← 0
7    p[s] ← NIL
8    Q ← s
9    while Q ≠ ∅
10       do u ← first[Q]          \\ u ist erstes Element in Q
11          for v ∈ Adj[u]        \\Nachbarn von u
12              do if color[v] = WHITE
13                  then color[v] ← GRAY
14                       d[v] ← d[u] + 1
15                       p[v] ← u
16                       ENQUEUE(Q, v)      \\ füge v in Q ein
17          DEQUEUE(Q)                      \\ lösche erstes Element aus Q
18          color[u] ← BLACK
```

Komplexität Die Initialisierung der Arrays dauert insgesamt $O(|V|)$ (Zeile 1 - 4). Die Operationen auf der Liste (Einfügen und Löschen) und den Arrays brauchen konstante Zeit, insgesamt $O(|V|)$. Das Durchsuchen der Adjazenzliste dauert $O(|E|)$. Damit beträgt die Gesamtkomplexität $O(|V| + |E|)$.

Klassifizierung der Kanten Beim ungerichteten Graphen unterscheidet man zwischen zwei Kantentypen:

- **Baumkanten (tree edges)**: diese Kanten führen zu unbesuchten Knoten und spannen den Graphen auf

- **Seitwärtskanten (cross edges)**: diese Kanten führen zu bereits besuchten Knoten

Beim gerichteten Graphen unterscheidet man zwischen drei Kantentypen:

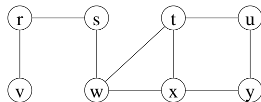
- **Baumkanten (tree edges)**: diese Kanten führen zu unbesuchten Knoten
- **Seitwärtskanten (cross edges)**: Kanten zu bereits besuchten Knoten
- **Rückkanten (back edges)**: Kanten zu Knoten in niedrigen Leveln (Zyklus im Baum)

BFS-Baum Die Breitensuche konstruiert einen Baum, der die Zusammenhangskomponente des Startknotens aufspannt. Der Weg im Baum vom Startknoten (Wurzel) zu den Nachfolgern entspricht dem kürzesten Weg bzgl. der Kantenanzahl im Graphen. Die Levelnummer eines Knotens entspricht der Höhe im Baum.

Anwendung Die Breitensuche ist besonders geeignet zur Lösung von Distanzproblemen. In ungewichteten Graphen liefert sie den kürzesten Weg bzgl. der Kantenanzahl vom Startknoten zu allen anderen Knoten, im gewichteten Graphen kann sie zum Dijkstra-Algorithmus erweitert werden. Die Unterscheidung der Kanten ermöglicht zum einen Zyklen zu finden, zum anderen erhält man einen Spannbaum der Zusammenhangskomponente des Startknotens.

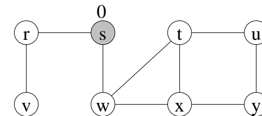
Beispiel zur Breitensuche im ungerichteten Graphen:

Schritt 0



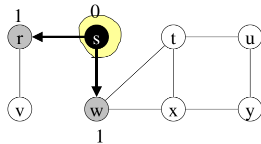
- für alle v ∈ V:
  color[v] = WHITE
  d[v] = ∞
  p[v] = nil

Schritt 1



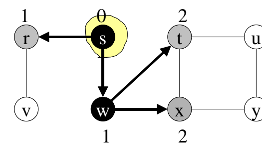
- Startknoten s wird grau markiert
- Q = (s)
- d[s] = 0

Schritt 2



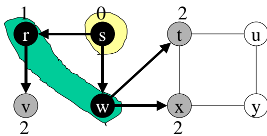
- tree edge von s zu r und w
- Q = (w, r)
- s ← schwarz
- Level 0 abgearbeitet

Schritt 3



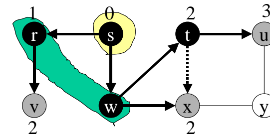
- tree edges von w zu t und x
- Q = (r, t, x)

Schritt 4



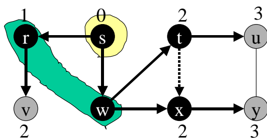
- tree edge von r zu v
- r ← schwarz
- Level 1 abgearbeitet
- Q = (t, x, v)

Schritt 5



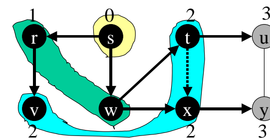
- tree edge von t zu u
- cross edge von t zu x
- Q = (x, v, u)

Schritt 6



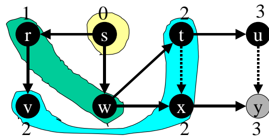
- tree edge von x zu y
- x ← schwarz
- Q = (v, u, y)

Schritt 7



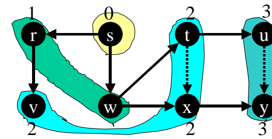
- v ← schwarz
- Level 2 abgearbeitet
- Q = (u, y)

Schritt 8



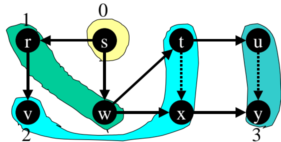
- cross edge von u zu y
- u ← schwarz
- Q = (y)

Schritt 9



- y ← schwarz
- Level 3 abgearbeitet
- Q = ( ) → Suche beendet

Schritt 10



- wellenartiges Ausbreiten
- tree edges: Level i zu i+1
- cross edges: bleiben im gleichen Level oder führen ins nächsthöhere Level

3 Tiefensuche

Im Gegensatz zur Breitensuche bewegt man sich bei der Tiefensuche (depth-first search) möglichst weit vom Startknoten weg, bevor man die restlichen Knoten besucht. Trifft man auf einen Knoten, der keine unbesuchten Nachbarn mehr hat, so erfolgt backtracking, d.h. die Suche wird beim Vorgänger fortgesetzt. Dadurch werden alle vom Startknoten aus erreichbaren Knoten gefunden.

Algorithmus Als Eingabe benötigt der Algorithmus einen Graphen und einen Startknoten. Wie bei der Breitensuche, werden auch bei der Tiefensuche Informationen zu jedem Knoten gespeichert:

- **color**[v]: repräsentiert den aktuellen Bearbeitungsstatus:
  weiß = unbesucht/unbearbeitet
  schwarz = abgearbeitet (v und alle Nachbarn von v wurden besucht)
  grau = in Bearbeitung (v wurde besucht, kann aber noch unbesuchte

Nachbarn haben)

- **p[v]**: Vorgänger (=predecessor) von v
- **b[v]**: Beginn der Suche (Einfügen des Knotens in den Stack bzw. Zeitpunkt des rekursiven Aufrufs)
- **f[v]**: Ende der Suche (Löschen des Knotens aus dem Stack bzw. Ende des rekursiven Aufrufs)

Die Knoten, die in Bearbeitung sind, werden in einem Stack K (LIFO) verwaltet.

DFS(G, s)

```
1   for each vertex u ∈ V[G]-{s}
2         do color[u] ← WHITE
3            b[u] ← ∞
4            f[u] ← ∞
5            p[u] ← NIL
6   time ← 1
7   color[s] ← GRAY
8   PUSH(K, s)
9   b[s] ← time
10  p[s] ← NIL
11  while K ≠ ∅
12        do u ← TOP(K)
13           if ∃v ∈ Adj[u] : color[v] = WHITE
14              then color[v] ← GRAY
15                   PUSH(K, v)
16                   b[v] ← time ← time + 1
17           else POP(K)
18              color[u] ← BLACK
19              f[u] ← time ← time + 1
```

Komplexität Das Initialisieren des Graphen dauert $O(|V|)$, Zugriffe auf den Stack und die Arrays brauchen konstante Zeit (insgesamt $O(|V|)$), die Adjazenzliste wird genau einmal durchlaufen ($O(|E|)$. Damit ergibt sich eine Gesamtlaufzeit von $O(|V| + |E|)$.

Klassifizierung der Kanten Beim ungerichteten Graphen unterscheidet man zwischen zwei Kantensorten:

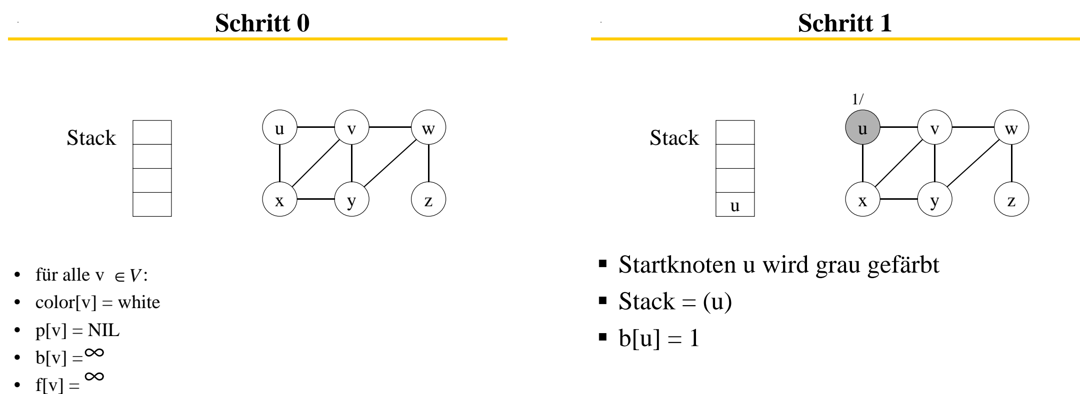
- **Baumkanten (tree edges)**: diese Kanten führen zu unbesuchten Knoten
- **Rückkanten (back edges)**: diese Kanten führen zu besuchten Knoten (Zyklus)

Beim gerichteten Graphen unterscheidet man zusätzlich noch zwischen

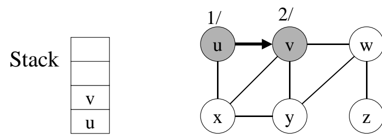
- **Vorwärtskanten (forward edges)**: Kanten zu Nachfolgern im DFS-Baum
- **Seitwärtskanten (cross edges)**: restliche Kanten

Anwendungen der Tiefensuche Der Algorithmus der Tiefensuche entdeckt alle Knoten, die vom Startknoten aus erreichbar sind. Diese Eigenschaft wird u.a. für die Garbage Collection in Java verwendet (mark-sweep). Im „mark“ -Teil werden mittels Tiefensuche alle erreichbaren Objekte durch einen Bitstring markiert, im „sweep“-Teil werden alle nicht markierten Objekte gelöscht. Eine weitere Anwendung der Tiefesuche ist die Bestimmung der starken Zusammenhangskomponenten. Dabei wird zunächst die Tiefensuche auf dem gerichteten Graphen ausgeführt, anschließend auf dem transponierten Graphen beginnend bei den Knoten mit den höchsten f-Werten. Die starken Zusammenhangskomponenten entsprechen den DFS-Bäumen des transponierten Graphen.

Beispiel zur Tiefensuche im ungerichteten Graphen

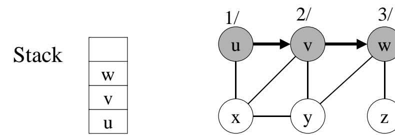


Schritt 2



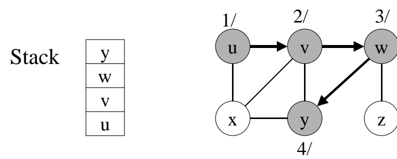
- tree edge von u zu v
- Stack = (v, u)
- b[v] = 2

Schritt 3



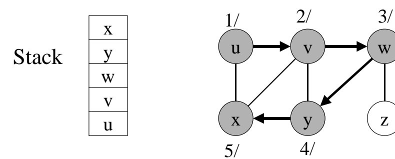
- tree edge von v zu w
- Stack = (w, v, u)
- b[w] = 3

Schritt 4



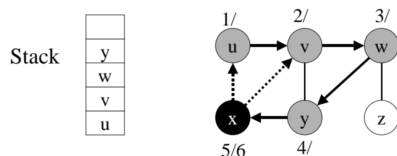
- tree edge von w zu y
- Stack = (y, w, v, u)
- b[y] = 4

Schritt 5



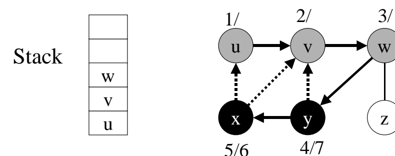
- tree edge von y zu x
- Stack = (x, y, w, v, u)
- b[x] = 5

Schritt 6



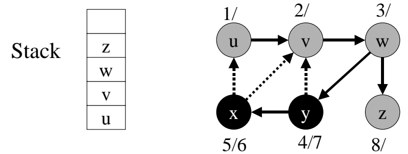
- back edge von x zu u und v
- Stack = (y, w, v, u)
- x ← schwarz
- f[x] = 6

Schritt 7



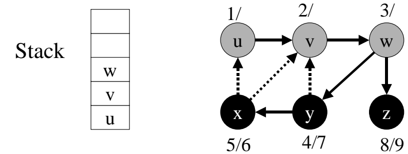
- backtracking zu y
- cross edge von y zu v
- Stack = (w, v, u)
- y ← schwarz
- f[y] = 7

Schritt 8



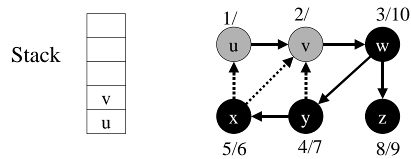
- backtracking zu w
- tree edge von w zu z
- Stack = (z, w, v, u)
- f[z] = 8

Schritt 9



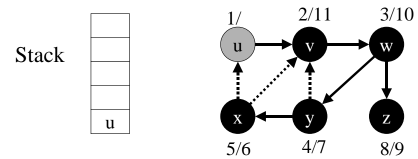
- Stack = (w, v, u)
- z ← schwarz
- f[z] = 9

Schritt 10



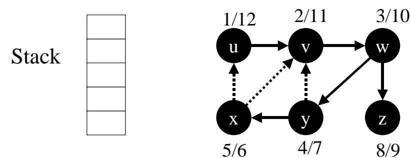
- backtracking zu w
- Stack = (v, u)
- w ← schwarz
- f[w] = 10

Schritt 11



- backtracking zu v
- Stack = (u)
- v ← schwarz
- f[v] = 11

Schritt 12



- backtracking zu w
- u ← schwarz
- f[u] = 12
- Stack = () → Suche beendet

4 Literaturliste

- T. H. Cormen; C. E. Leiserson; R. L. Rivest: Introduction to Algorithms The MIT Press, Cambridge, Mass. u.a.,S. 469ff

- M. T. Goodrich, R. Tamassia: Data Structures and Algorithms in Java, John Wiley and Sons, Inc., 1998. S 557ff

- T. Ottmann; P. Widmayer: Algorithmen und Datenstrukturen, Spektrum Verlag Heidelberg, S. 551ff